# Perspectives on the IRIS Explorer Visualization Environment

Al Globus, Computer Sciences Corporation

Report RNR-92-021

18 May 1992

This work was supported by NASA Contract NAS 2-12961 to Computer Sciences Corporation for the Numerical Aerodynamic Simulation Systems Division at NASA Ames Research Center.

*Copies of this report are available from:*

NAS Applied Research Office
Mail Stop T045-1
NASA Ames Research Center
Moffett Field, CA 94035
(415) 604-4332

# Perspectives on the IRIS Explorer Visualization Environment

A. Globus, Computer Sciences Corporation[1]

*Abstract*

*A critique of Silicon Graphics, Inc.'s medium grain data flow scientific visualization environment IRIS Explorer is presented. Explorer shows promise as an important tool for users to quickly and easily build custom distributed visualization applications. Developers can use Explorer to create new visualization techniques without re-implementing common functionality. There are, however, significant flaws. Although Explorer is no worse than much new UNIX software, by absolute standards Explorer is buggy. Explorer also uses a bit too much memory and is too slow for highly interactive tracking tasks. These problems should be solvable. Database management is weak and this is inherent in the data flow model. The flaws notwithstanding, Explorer is sufficiently reliable and capable to do a great deal of useful work.*

## Introduction

This paper is based on using IRIS Explorer 1.0 [SGI92a] at NAS[2] for six months developing new computational fluid dynamics (CFD) visualization techniques. Although there were many frustrating moments along they way, we were able to visualize a number of data sets and develop and test new visualization algorithms.

Explorer is a general purpose scientific visualization software environment bundled with Silicon Graphics, Inc. workstations. Figure 1 is a screen dump of an Explorer working session. Explorer is available on the Cray YMP and is being ported by other hardware vendors. Explorer uses a WIMP[3] user interface to data flow programming; an approach pioneered by Upson89. A well designed traditional application will usually outperform Explorer for a particular task, but an investigator frequently needs something slightly different than what any given application does. Data flow programming combined with a suitable set of processing modules provides enormous flexibility. Subroutines written in C, C++ or FORTRAN can, in many cases, be easily integrated into Explorer.

Explorer has its own terminology for data flow. The data flow diagrams are called *maps*. Data

1. This work is supported through NASA contract NAS2-12961.

2. The NAS (Numerical Aerodynamic Simulation) Systems Division at NASA Ames Research Center is a supercomputer facility for the study of computational aerosciences. NAS has two large Crays, a Connection Machine, an Intel hypercube, a large Convex, 150+ Silicon Graphics Inc. workstations, a number of Suns, and a multi-terabyte mass storage system communicating via ethernet and high speed (ULTRA) local networks. NAS also supports AERONET, a wide area network providing NAS services to industry, government, and academia.

3. Windows, Icons, Mice, and Pointing

flow diagrams consist of nodes connected by edges, similarly, Explorer maps consist of *modules* connected by *wires*. Modules process data and data flows along wires. Each module is implemented as a separate UNIX processes. Wires are implemented in shared memory where feasible and with UNIX pipes elsewhere; e.g., between machines or on a Cray.

Explorer focuses on visualization of static 2-D and 3-D scalar fields. Visualization of vector fields is weak and complex time series visualization problematic. The architecture to visualize vector fields exists although more modules are needed. For time series visualization one cannot easily control the database created or guarantee deterministic, synchronous generation and display of large time step visualizations in all cases.

Since the SGI marketing literature and manuals elucidate the positive aspects of Explorer, this paper will focus on deficiencies and problems. Although there are many such, we are generally pleased with Explorer for visualization of steady state data and use it frequently. One should note that the bugs mentioned here exist at the time of writing and may have been subsequently fixed. Finally, few deficiencies noted in this paper are unique to Explorer. They are shared, in most cases, by other scientific visualization environments. All performance figures in this paper were taken on a SGI 320 VGX with 64Mbytes of memory.

The remainder of this paper is divided into sections on user, programmer, and visualization researcher perspectives on IRIS Explorer.

## User Perspective

The user perspective includes sections on available modules, user interface, performance, reliability, importing data, save/restore, video production, distributed processing, time dependent visualization, and an analysis of Explorer as a programming language.

### Modules

A list of modules is maintained by the *Librarian*. These modules may be dragged into the *Map Editor* to become part of the current map. SGI provides a number of modules and more are in the public domain. There is a good selection of scalar field visualization modules, a wide variety of image processing modules, a couple of rendering modules, and a few handy utility modules. Utility modules include colormap generators, histograming, a data set generator, disk IO, subsampling, etc. The utility module collection is not complete, the author has found it necessary to write a number of simple utility modules to perform various mundane tasks. In addition, there is a paucity of vector field visualization modules. For example, we have yet to find a tangent curve module.

There is terrific unsupported module called RenderRemote. This provides a unique feature: rendering graphics on a remote machine. This allows one to put the widgets and map on an inexpensive X-server and devote an entire high performance graphics screen to the visualization. RenderRemote can also be used such that several physically separated individuals on the same network can view the same visualization. Nothing is perfect, and RenderRemote has a performance problem that can cause rotating objects to jerk every few seconds. This is merely a bug, not a fundamental problem with the design.

### User Interface

Explorer has WIMP user interfaces for all portions of the system. There is no command or scripting language for normal operation, although several are hidden beneath the surface.

The Apple Macintosh is recognized for its excellent user interface. Mac programs are supposed to conform to a set of guidelines [Apple85]. According to these guidelines, the user should feel in control of the system, not the other way around. This is achieved by systems that embody three qualities: responsiveness, permissiveness, and consistency. In addition, modes should be avoided or at least clearly indicated. Additionally, software systems should be forgiving, convenient, and self-explanatory. We examine the Explorer user interface in these terms.

- Explorer is somewhat responsive. Responsiveness is related to the directness of manipulation and speed of response. Response should be essentially instantaneous for discrete inputs (e.g., setting a parameter) and >8 frames per second for tracking tasks (e.g., manipulating viewpoint). One can directly manipulate the Explorer map. The data are indirectly manipulated by programming the map. Graphic output is both directly and indirectly manipulated. Visual feedback for actions usually meets the performance requirements, but semantic completion is frequently orders of magnitude slower. For example, dragging modules is fast and smooth but it takes a few seconds for module to launch, manipulating widgets is quick but widgets only output numbers when the mouse button is released, and the simplest animations on small data sets, e.g. sweeping through grid planes, is far slower than 8 frames a second on our 4D 320 VGX.
- Explorer is very permissive. One can assemble modules in any combination and connect any two modules where inputs and outputs are type compatible. This is true even if modules run on many heterogeneous machines.
- Explorer is fairly consistent. All modules present the same user interface and all module widgets are similar. However, other portions of the system do not exhibit such consistency. For example, there are multiple file browsers each with a different layout and behavior.
- Explorer is relatively modeless. There are no hidden modes or switches that produce radically different behavior from apparently similar objects, except in the Render module. The render module has multiple navigation modes.
- Explorer in not very forgiving. There is no undo, the most important forgiveness mechanism. If a module corrupts shared memory, which happens from time to time, one must exit Explorer and restart.
- Explorer is not particularly convenient. The data flow model requires a great deal of manipulation to do simple visualizations. The Librarian is clumsy. As mentioned before, widgets only take effect when released so it is tedious to interactively hone in on an unknown value.
- Once one understands the basics, Explorer is largely self-explanatory. Each module has on-line help which is almost always very useful. The development tools make on-line help production very easy (see below) so that help files are usually quite good.

**Performance**

The worst performance problem with early data flow visualization systems was excessive copying of data. Wires were implemented as UNIX pipes so data were multiply copied for each wire. This is prohibitive for large data sets. Explorer has solved this problem on SGI workstations by using

shared memory to implement data flow and avoiding unnecessary copy operations. The Cray has no shared memory, so data passed between modules will be copied. This can be a significant overhead.

Like any general purpose tool, efficiency will suffer compared with more special purpose implementations. For example, to loop through grid planes displaying a scalar field from a blunt fin data set[4] [Hung85] where the data required 7.2 Mbytes of storage, FAST [Bancroft90] required 12.4 Mbytes and Explorer required 14.3 Mbytes, a 40% increase in memory overhead. FAST was noticeably faster than Explorer for this task.

It is possible to perform some direct data manipulations using a map including the PickLat module, but even with small data sets interactive performance is unacceptable. We created a map using PickLat to select a small portion of a 12 x 12 x 12 uniform data set. The portion chosen was averaged and this average was used as an isosurface threshold. Even with this tiny data set, delays between mouse movements and isosurface drawing were on the order of seconds.

The performance of the Render module is excellent. Point of view changes are very fast, even with many polygons requiring lighting, transparency, etc. However, when there is enough graphics data, performance eventually degrades. One can specify wireframe or points-only drawing to improve performance, but there is no adaptive degradation mode to keep update rate constant as the amount of graphics data increases.

### Reliability

Explorer 1.0 has many obvious bugs. When exploring a new aspect of the system, we inevitably runs into more. Bugs are a problem from two perspectives. First, they are frustrating to work around. More important for scientific purposes, they put results into question. If there are numerous obvious bugs, one suspects even more numerous subtle bugs; bugs that produce slightly incorrect results. Thus, confidence in the results produced by Explorer is undermined; or should be.

To increase confidence in results, one could use two modules that should produce similar results. These modules need to be developed independently. If the module outputs are equivalent for identical input, the probability that the both are incorrect is the small since the total probability is the product of the probability of each being in error, numbers that are generally much less than one.

### Importing Data

One of the tedious tasks in scientific visualization is converting data formatted by sensors or numerical simulations into a form that visualization software can understand. To address this issue, SGI provides a WIMP application called *Data Scribe*. Data Scribe creates modules that read disk files into Explorer lattices[5]. The WIMP interface is used to describe the format of the file and how the data should be shuffled into lattice format. Data Scribe is sufficiently general that most semi-reasonable data formats equivalent to an n-dimensional array can be read correctly. We have not used Data Scribe a great deal as C++ modules are faster, but others have given Data Scribe rave reviews.

### Save and Restore

Maps can be saved to disk and read back in. For the most part this works well and is of great util-

4. 40x32x32 nodes on a curvilinear grid
5. A lattice is the Explorer multidimensional array data structure.

ity. Unfortunately, modules do not save internal state. This deficiency is serious for the Generate-Colormap and Render modules. These modules have a complex internal state which is lost when a map is saved to disk. There is also a serious bug. When modules are combined into a group (see below), they may be saved to disk but cannot be read back in. This substantially limits the utility of module grouping.

All the Explorer data structures may be saved and restored in ascii or binary formats. This format is not documented nor does SGI promise stability of the format in future releases, although backwards compatibility may be expected.

### Video

There are two serious problems when using Explorer to produce scientific videos. Both could be easily corrected by the Explorer development team but cannot be worked around by current users.

There is no way control line pixel width in Explorer. If a module sets the color of a Geometry object, and several do, the Render module cannot over-ride with another color. This may cause problems when producing videos.

There is no way to use stop frame video production techniques in Explorer. Thus, if performance is not sufficient to take video off the screen in real time there is a problem. Since many excellent scientific visualization videos are created using stop frame techniques, its absence is a major deficiency.

### Distributed Processing

Explorer is designed such that modules on multiple machines can work together. This works very well. One simply brings up a Librarian for each machine and drags modules onto the Map Editor in the usual way. Figure 2 is a distributed Explorer map. Format conversions are automatic[6] and

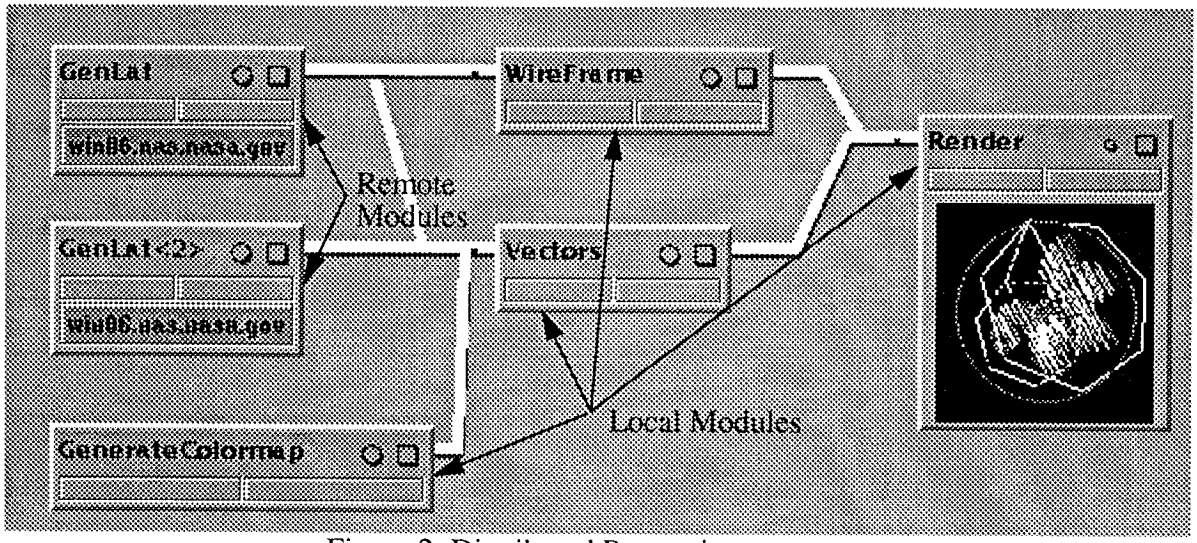


Figure 2: Distributed Processing

all networking issues, except timing, are transparent. There are a couple of performance problems that are very easy to work around and one that requires the SGI's attention.

---

6. Except for the 'Unknown' data type, since Explorer does not know anything about its internal structure. See below for a discussion of data types.

When two modules are connected, the wire appears briefly then disappears. After a delay, it re-appears. When the two modules are on different machines, the delay can be lengthy. When the wire disappears, one justifiably suspects that the connection failed, although this is usually not the case.

When passing a single data item over several wires that pass between two machines, the same data is sent over each wire. I.e., the same data is transferred several times. This can be avoided by passing the data to a single module on the remote machine then connecting multiple wires from that module to the data's destinations. SGI does not provide such a module but it is trivial to write.

There is one performance problem for which no easy work around exists. When Explorer sends data over a network the data are converted to network neutral format for the send and converted to local format when received. If the sending and receiving machines are compatible, this conversion is unnecessary.

**Time Dependent Visualization**

Very simple time dependent visualization is possible with Explorer, but more complex visualization involving multiple geometries or large data sets is difficult or impossible. Time dependent visualization minimally requires looping, convenient control of time step, and deterministic synchronous presentation of time steps. As we will see, looping can be accomplished. In addition, time step can be easily controlled with a widget. Thus a single Geometry can be produced per time step and sequenced properly (most of the time), but when multiple geometries must be coordinated problems arise.

Visualizations from different time steps may become mixed into the same picture. In the general case, there is no way to insure that each frame produced by Render contains Geometries from only one time step. This can happen because Explorer data have no concept of where they come from or where they are going. Thus, if time steps are input to a map continuously and several Geometries are produced per time step, each Geometry will be produced asynchronously. As long as there are no missing time steps for any Geometry this should work. If one of the Geometry streams misses a step for some reason, then the output will subsequently contain Geometries from different time steps.

There is at least one way to lose time steps. This can be caused by Explorer's lack of input port queueing. If the situation in figure 3 exists, data entering *input* module can be overwritten if the *data path* takes long enough that the *time step source* module produces the next time step before *data path* completes.

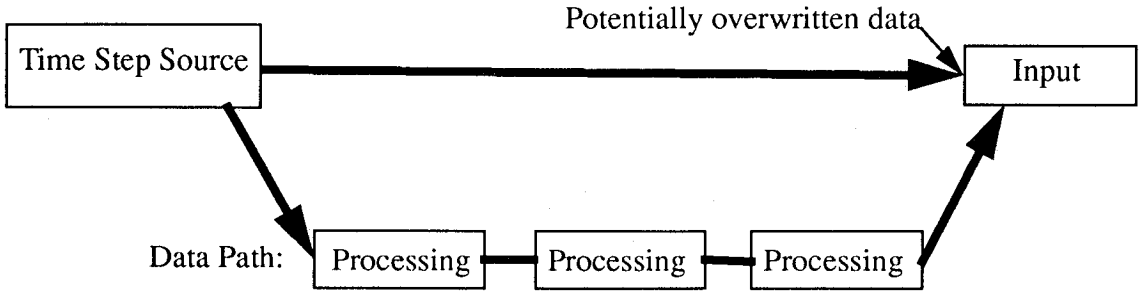


Figure 3: Lack of Input Queueing Problem

Even if the Figure 3 situation does not exist, the Render module cannot reliably present all time

steps. The Render module only displays when it is not busy. If one sends Geometry objects to the Render module fast enough, it spends all its time updating data structures and may not draw a particular time step at all! This deficiency can also causes problems when sweeping planes or isosurface levels through data.

### Language Issues

Explorer is programmed with a graphical language. Thus, one may ask what programming language features Explorer provides. Programming languages generally have mechanisms for sequencing, looping, selection, and subroutine calls. All of these are provided or can be easily built in Explorer.

Sequencing is straightforward. Simply connect a series of modules together with wires from one to the next.

Looping is harder but is frequently useful in scientific visualization. For example, looping through a set of parallel cutting planes to visualize a volume. Looping can be difficult in data flow languages. SGI provides one limited loop module (Timer) and we had little difficulty writing modules to implement other kinds of looping. Since loops generally involve several modules and each is a separate process, performance is much worse than for loops built into a traditional application.

Selection modules (If-Then-Else, Case, etc.) are not provided but we wrote a simple If-Then module in a few minutes.

Subroutine call functionality is provided by grouping modules into a single new module. WIMP tools are provided to choose a subset of the possible inputs and outputs for the interface to the new module -- and to rename them if desired. There is also a WIMP tool to create a custom control panel for the group. The group feature will prove to be very valuable once the save/restore bug mentioned earlier is fixed.

## Programmer Perspective

Explorer's capabilities may be extended by developing new modules [SGI92b]. New modules may be created in one of two ways: using the Module Builder to convert a C, C++, or Fortran subroutine into a module, or writing a Shape program for the LatFunction module. In general, adding modules to Explorer is quite easy and involves very little programmer time beyond that needed to code the algorithm involved.

### Module Building

The Module Builder is a WIMP application that creates code to interface user subroutines to Explorer. One specifies the inputs, outputs, and a control panel. The C, C++ or FORTRAN subroutine's arguments and return values are also defined. Finally, the connections between the input/output ports and the subroutine's arguments and return value are defined. This is all very easy and fairly intuitive.

Building a widget based control panel for a module is trivial. The Module Builder provides a WIMP interface to choose parameters to control, type of widget to use, and for widget layout on the control panel. Control panel layout is via direct manipulation. If a more sophisticated user interface is desired, there are Explorer widgets that create X and/or GL windows.

It is possible to generate compile or run time errors with the Module Builder by incorrect specifi-

cations or connections, or by entering code scraps that contain errors. When they occur, such bugs can be hard to find since there is no debugger that directly relates errors to parts of the Module Builder user interface. Also, one must click though a fairly extensive hierarchy of windows to look at everything.

The Module Builder not only generates code, it manages all of the files and resources to fully integrate new modules into Explorer; including automatically generated documentation files for on line help and man pages. The developer adds explanatory text to the documentation files. When module inputs/outputs change and the automatic documentation must be rewritten, the developer added text is properly preserved.

### Data Structures

It is possible to convert subroutines that know nothing about Explorer data structures into Explorer modules. One can use the Module Builder to extract necessary data from input ports, pass it in subroutine arguments, and get return values to output ports. However, the developer will still need to understand Explorer data structures in order to use the Module Builder correctly. Furthermore, it is frequently easier to operate on Explorer data structures as wholes rather than in pieces. For example, our C++ classes have contructors with Explorer data types as arguments.

Explorer uses five data structures: parameter, lattice, pyramid, geometry and unknown. There is no class hierarchy relationship.

Parameters are single values such as floating point numbers, strings or integers.

Lattices are multidimensional arrays with coordinate systems. There are two parts to a lattice: data and coordinates. The data are simply a multi-dimensional array stored in FORTRAN order. The coordinates can be 1, 2, or 3-D and uniform (regular), perimeter (stretched), or curvilinear. Lattices are required to have data so lattices containing only curvilinear coordinates cannot be generated when only the grid is of interest. This is unfortunate as particle trace, grid generation, and other modules might prefer to have only coordinates.

Pyramids are made up of a base lattice and a list of lattices and connections. The connections specify relationships between nodes in two lattices. Pyramids are interpreted by SGI modules to be finite elements (unstructured grids) or molecules. Pyramids may be interpreted differently by other sets of modules. In particular, pyramids could be used for multi-zone, iblanked data sets [Buning89] (see below), but we have not written any Explorer modules to process multi-zoned or iblanked data.

Geometries are rendered by SGI provided graphics modules. SGI also supplies modules to convert Lattices and Pyramids to Geometry. Geometry data structures are designed for fast rendering using the SGI GL. Geometry is the same data structures used by the SGI 3-D toolkit software (Inventor, formerly Scenario).

Unknown is an array of uninterpeted bytes. It can be used in any way a set of co-operating modules desire.

### Memory management

Memory management can be difficult. Memory intensive items, such as data and curvilinear coordinates, are kept in a data structure that include a reference count. Whenever a module inputs such an object, the reference count is incremented. When a module no longer needs an object the reference count is decremented. When the count indicates the data is no longer needed, it is freed. In

principle this should work fairly well. However, there are no tools to examine the state of shared memory used for inter-module communication on an SGI workstation. There are no tools, other than a debugger, for examining reference counts. Thus, memory leaks and other memory management pathologies are difficult to detect. Not only is it difficult to write memory-leak free modules of any complexity, it's quite possible that SGI's modules leak memory as well. Whenever we use Explorer for any length of time, our 64Mbytes of memory inevitably fill up. This may or may not indicate a memory leak and there's no easy way to find out.

Data on input or output ports is not freed until replaced by new data or the module is destroyed. There is no convenient way to delete this data without destroying the module. One could send a NULL to input ports so that the data is deleted but this doesn't work for unwanted data on output ports. Worse, there is no visual indication that a wire, input port, or output port holds data.

### Type conversion

Modules may be written to operate on a single type, e.g., float or int, and still operate correctly on inputs consisting of bytes, shorts, longs, floats, or doubles. This is accomplished by using the Module Builder to force the conversion. The Module Builder can write code to convert the data to the proper type and pass the new data to one's subroutine. This functionality is not available, however, if a subroutine interfaces to entire lattices or pyramids. Type conversion is only accomplished when the Module Builder is used to break up lattices and pyramids into arrays and scalars.

### API[7]

Explorer comes with a fairly conventional subroutine library API. This library gives the programmer control over all Explorer objects relevant to modules. The library is reasonably well documented and provides complete access for C and FORTRAN, except that input/output port management is not supported from FORTRAN. There is no specific C++ interface which although one can use the C interface.

### Networking

When writing modules, one need not consider networking issues unless using the Unknown data type. All other data types are automatically and - as far as our testing reveals - correctly converted as needed.

### LatFunction

SGI provides the LatFunction[8] module to operate on multi-dimensional arrays (i.e., lattices). LatFunction implements an interpreted language with a C-like syntax called Shape. Shape is intended to quickly prototype computational modules; i.e., to convert a set of input lattices and parameters into a set of output lattices. Learning Shape is somewhat difficult and the error messages are of little utility since there are no line numbers. Once mastered, however, LatFunction is incredibly useful. One can quickly try most ideas that can be expressed as data parallel manipulations of lattices. In addition, many utility modules can be easily implemented.

LatFunction can be used to build Explorer modules with lattice and parameter inputs, lattice outputs, and custom widget panels using the Module Builder. Parameter outputs would be very useful but are not provided. We use LatFunction frequently.

---

7. Application Programmer Interface.
8. Note: the version of LatFunction shipped with Explorer 1.0 has a very serious bug. However, LatFunction2 which was distributed at the SGI Developer's Forum fixed this bug.

### Multi-Zone Iblank

The format produced by the largest most complicated simulations at NAS, Plot3d multi-zone, iblank[9] (Buning89), is not directly supported by Explorer. It is possible to adapt the Explorer pyramid data structure to implement visualization techniques on these data, although we have not done so. Every module manipulating lattices would have to be modified or rewritten to input the new format.

One approach to using pyramids for multi-zone iblank support uses a pyramid's base lattice for a descriptive string so that the data structure has self knowledge. The grid and field zones are then stored in the pyramid's other lattices, one zone per lattice. Each grid zone is held in a separate lattice with the data portion used for the iblank array. Field zones each have a separate lattice, the coordinate portion may or may not contain the grid point location. Duplicating the grid posses no major space penalty as only the pointers need be duplicated, not the grid data. Information on inter-zone topology can be stored in additional lattices. The format may be extended by adding syntax to the description string. The string should use LISP syntax to avoid inventing yet-another-interpreted-language-without-a-debugger [Hultquist92].

## Visualization Research Perspective

Within the limitations imposed by the data flow model, Explorer shines as a visualization research tool. The investigator need pay only cursory attention to secondary issues and focus almost exclusively on the visualization problem at hand. Furthermore, it is trivial to compare results with traditional visualization techniques for which modules exist. We completed a simple research project, Gridigrator [Globus92a], including pictures and a paper (but no video), in two weeks. This included the time to learn Explorer and C++. Learning C++ was the single most time consuming aspect of the work.

LatFunction (see above) provides a marvelous environment for quickly testing out many visualization ideas. LatFunction could have been used to implement Gridigrator. One can quickly and easily write and test programs to perform a wide variety of array manipulations. The author has had a great deal of success trying ideas in minutes that might have taken hours or even days if the software were implemented in C or C++.

## Database Management

In the data flow model the data are implicit and hidden; data exist in a vague way on the wires, input ports, and output ports. Lang91 and Globus92b argue that database management is central to scientific visualization, although it is usually treated as a peripheral issue. Many of the memory management problems discussed above stem from lack of database management. Finally, it is difficult to control what data is saved or discarded by Explorer, a critical issue when visualizing massive data sets [Globus92b]. We see no solution to the problem of inadequate database facilities, it appears to be inherent in the data flow model.

## Conclusion

Explorer 1.0 is useful for visualization of simple static scalar fields. The user interface is quite

9. Iblank refers to an integer mask indicating meaningless data points.

reasonable. Some key scientific visualization issues, such as flexibility and unique data format input, have been directly and successfully addressed. On the other hand, performance and reliability need substantial improvement.

Improvements in functionality and scope can be expected because module building is easy and Explorer is bundled with SGI workstations providing a large user base. The Module Builder makes writing modules for Explorer an activity dominated by coding an application, not dealing with obscure interface issues. When developing most modules, user interface, type conversion, and networking issues are easily dealt with.

On the other hand, memory management can be difficult. The greatest structural weakness of Explorer is the lack of an explicit database model. This is common to all data flow systems. Unfortunately, to a great degree scientific visualization consists of building a database of objects that are viewed in various ways. Explorer provides only indirect access to these objects.

Multi-zone, iblank data are important for CFD research at NAS, but are not directly supported by Explorer. An adaptation of Explorer data structures can address this issue. Modules to implement common visualization techniques on these data structures must then be developed.

Explorer appears poorly suited for time dependent visualization except in very simple cases. It may be possible to write modules that will allow reasonable time varying visualization if adequate storage is not an issue, but the resulting maps may be error prone.

In general, we have found IRIS Explorer to be useful if somewhat limited. We expect great improvements in the future, particularly in the number of modules available.

## Acknowledgments

The SGI Explorer development has been extraordinarily helpful and responsive in assisting with problems we've had, fixing problems we pointed out, and discussing Explorer's strength and weaknesses. E. Raible and A. Vaziri reviewed drafts of this paper and made many helpful comments.

## References

[Apple85] *Inside Macintosh, Volume 1*, Addison-Wesley Publishing Company, Inc., ISBN 0-201-17731-5.

[Bancroft90] G. Bancroft, F. Merritt, T. Plessel, P. Kelaita, K. McCabe, A. Globus, "FAST: A Multi-Processing Environment for Visualization of CFD," *Proceedings Visualization '90*, IEEE Computer Society, San Francisco (1990).

[Buning89] P. P. Walatka, P. G. Buning, *PLOT3D User's Manual,* NASA Technical Memorandum 101067, NASA Ames Research Center.

[Globus92a] A. Globus, "Gridigrator: A Very Fast Volume Renderer for 3D Scalar Fields Defined on Curvilinear Grids," NASA Ames Research Center, NAS Systems Division, Applied Research Branch technical report RNR-92-001, January 1992.

[Globus92b] A. Globus, "A Software Model for Fast Flexible Visualization of the Largest Time Dependent Volumetric Numerical Simulations," NASA Ames Research Center, NAS Systems Division, Applied Research Branch technical report expected in 1992.

[Hultquist92] J. P. M. Hultquist and E. L. Raible, "SuperGlue: A Programming Environment for

Scientific Visualization," NASA Ames Research Center, NAS Systems Division, Applied Research Branch technical report RNR-92-014, April 1992.

[Hung85] C.H. Hung, P.G. Buning, "Simulation of Blunt-Fin-Induced Shock-Wave and Turbulent Boundary-Layer Interaction," *J. Fluid Mech.* (1985), Col. 154, pp. 163-185.

[Lang91] U. Lang, R. Lang, and R. Ruehle, "Integration of Visualization and Scientific Calculation in a Software System," *Proceedings IEEE/ACM SIGGRAPH Visualization '91*, 22-25 October, 1991, San Diego, California.

[SGI92a] *IRIS Explorer User's Guide*, Silicon Graphics Computer Systems, Document 007-1371-010, 1992.

[SGI92b] *IRIS Explorer Module Writer's Guide*, Silicon Graphics Computer Systems, Document 007-1369-010, 1992.

[Upson89] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, A. van Dam, "The Application Visualization System: A Computational Environment for Scientific Visualization," *IEEE Computer Graphics and Applications*, July 1989, pp. 30-41.

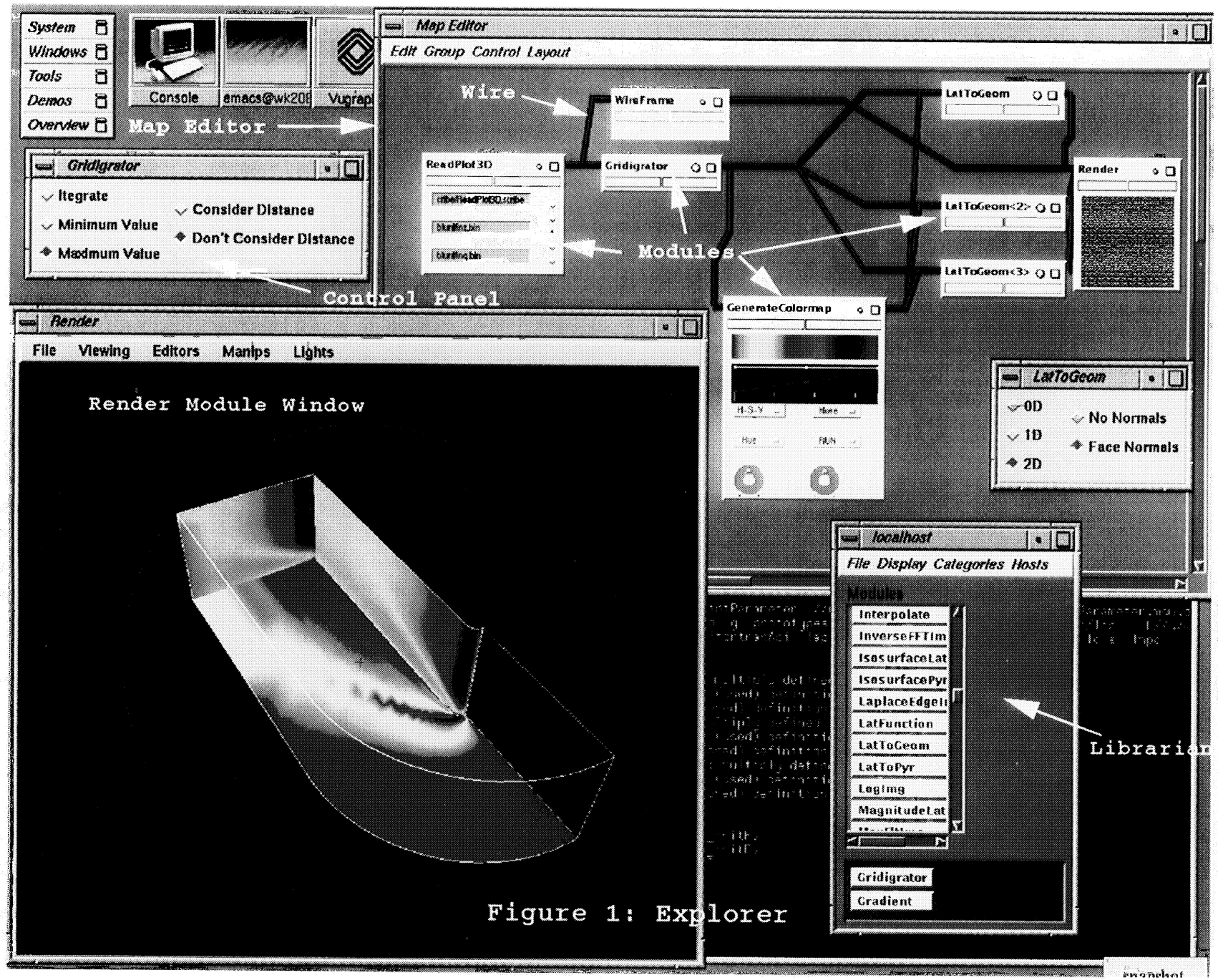


Figure 1: Explorer